Accelerating Graph Algorithms Using Graphics Processors: Shortest Paths for Planar Graphs

**Hristo Djidjev,
Sunil Thulasidasan, CCS-3;
Guillaume Chapuis,
Rumen Andonov, University
of Rennes, France**

We present a new approach to solving the shortest-path problem for planar graphs. This approach exploits the massive on-chip parallelism available in today's Graphics Processing Units (GPU). By using the properties of planarity, we apply a divide-and-conquer approach that enables us to exploit the hundreds of arithmetic units of the GPU simultaneously, resulting in more than an order of magnitude speed-up over the corresponding CPU version. This is part of our larger algorithmic work on finding efficient ways to parallelize unstructured problems, such as those found in complex networks and data mining, on highly parallel processors.

The shortest-path problem is a fundamental computer science problem with applications in diverse areas such as transportation, robotics, network routing, and Very Large Scale Integration (VLSI) design. The problem is to find paths of minimum weight between pairs of nodes in edge-weighted graphs, where the weight of a path p is defined as the sum of the weights of all edges of p. The distance between two nodes v and w is defined as the minimum cost of a path between v and w.

There are two basic versions of the shortest-path problem. In the single-source shortest-path (SSSP) version the goal is to find, given a source node s, all distances between s and the other nodes of the graph. In the all-pairs shortest-path (APSP) version, the goal is to compute the distances between all pairs of nodes of the graph. While the SSSP problem can be solved very efficiently in nearly linear time by using Dijkstra's algorithm [1], the APSP problem is much harder computationally. The fastest algorithm for general graphs is Floyd-Warshall's algorithm [1] that runs in $O(n^3)$ time and works for graphs with arbitrary (including negative) weights. That algorithm has a relatively regular structure that allows parallel implementations with high speedup. However, the cubic complexity of the algorithm makes it inapplicable to very large graphs.

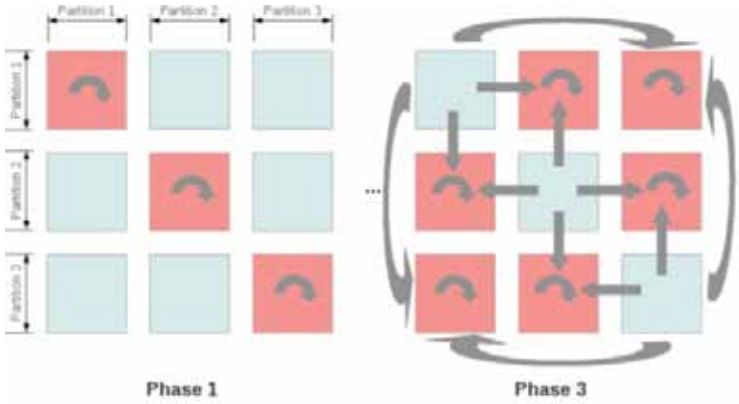

Fig. 1. Block-level data parallelism and data dependencies in the adjacency matrix for phase 1 and phase 3 of the algorithm. Sub-matrices for which computations are required are shown in red. Arrows indicate data dependencies.

As part of our work on solving unstructured graph-based problems on fine-grained parallel architectures, we describe a new algorithm for the APSP problem for planar graphs based on the Floyd-Warshall algorithm. The complexity of our algorithm with respect to the number of nodes is close to quadratic, while its structure is regular enough to allow for an efficient parallel implementation that enables us to exploit the massive on-chip parallelism of GPUs.

Our algorithm uses the Floyd-Warshall algorithm as a sub-routine, which successively re-evaluates the path between nodes i and j, by considering the path through vertex k, for all possible k. The structure of the algorithm is similar to the one of matrix multiplication, that makes very regular efficient parallel implementation possible. Our algorithm decomposes the graph into p parts, solves the APSP problem for the sub-graph induced by each part (in parallel, if more than one processor is available), and then uses that information to compute the distances between pairs of arbitrary nodes. The details of the algorithm are given in [2].

The above algorithm was implemented on a GPU using CUDA, nVIDIA's parallel programming framework for the GPU. Modern GPUs are efficient at manipulating structured data like matrices, and their highly parallel architecture (a GPU trades the complicated cache and control logic in a CPU for a large number–often hundreds–of arithmetic units) makes them ideally suited for processing large blocks of data simultaneously (referred to as the SIMD–Single Instruction Multiple Data–paradigm). In order to comply with the GPU paradigm, we define a computational kernel that implements Floyd-Warshall's algorithm and computes the APSP over a sub-matrix of the initial matrix. We then define computation grids, where blocks correspond to sub-matrices that can be computed simultaneously.

Phase 1 of the algorithm consists of computing APSP in self-dependent (in terms of data dependencies) diagonal sub-matrices, using a

For more information contact Sunil Thulasidasan at sunil@lanl.gov.

Funding Acknowledgments
LANL, Laboratory Directed Research and Development Program

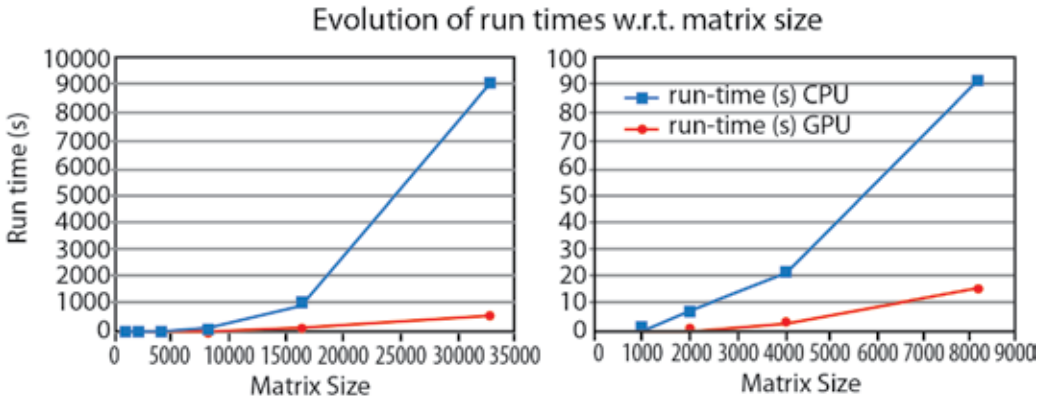

Fig. 2. Run times (left) with respect to input matrix size or the CPU and GPU versions. First four data points (right) zoomed in.

block-parallel version of the Floyd-Warshall algorithm. Computations for these diagonal sub-matrices can be run in parallel in different GPU blocks. Phase 2 of the algorithm consists of computing APSP on a sub-graph of the initial graph solely comprised of boundary vertices. For this purpose, we implemented a GPU version of the blocked APSP algorithm described in [5], a variant of the Floyd-Warshall algorithm where computations are divided into groups that can be easily be mapped to GPU blocks. Phase 3 of the algorithm consists of computing APSP in the remaining non-diagonal sub-matrices, using the same parallel Floyd-Warshall algorithm. Since Phases 1 and 3 use the same data patterns we illustrate their data dependencies in Fig. 1.

In order to test the efficiency of the algorithm, we compare two implementations of the partitioned APSP algorithm: (1) a single core CPU implementation of the partitioned all-pairs shortest-path algorithm (referred to as CPU version), and (2) a single GPU implementation of the partitioned all-pairs shortest-path algorithm (later referred to as GPU version).

The CPU version runs on an Intel(R) Xeon(R) CPU X5675 at 3.07GHz. The GPU version runs on an nVidia Tesla m2090 consisting of 512 cores at 1.3 GHz. The benchmark consists of random cost adjacency matrices, representing planar graphs with sizes ranging from 1024 vertices to 32,768 vertices. These graphs were generated using the LEDA graph generator [3]. Figure 2 shows the GPU versus CPU speed-up comparison for progressively larger graphs, with the GPU being up to 14 times faster than the GPU for larger graphs (32-k nodes). For the largest instance, the cost adjacency matrix requires about 4.2 GB of RAM. Instances larger than about 38,000 vertices would require more RAM than currently available on the GPU. One way around the memory size problem is to exploit the spatially constrained nature of paths in real-world graphs that will allow us to consider only a relatively small subset of the original graph [4]. We are also currently extending this work to tackle larger graph instances by using multi-GPU clusters.

[1] Cormen, T.H. et al., *Introduction to Algorithms,* 1st edition, MIT Press and McGraw-Hill, ISBN 0-262-03141-8 (1990).

[2] Djidjev, H.N. et al., "On Solving Shortest Path Problems for Planar Graphs using Graphics Processors," LA-UR-12-25700 (2012).

[3] Mehlhorn, K. et al., *LEDA: A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, (1999).

[4] Thulasidasan, S., "Heuristic Acceleration of Routing in Transportation Simulations Using GPUs," *Proceedings 4th International Conference on Simulation Tools and Technologies (SimuTools)* (2011).

[5] Venkatraman,G. et al., *J Exp Algorithmics*, **8**, 2.2 (2003).